An empirical analysis of source code metrics and smart contract resource consumption

Ajienka, Nemitari; Vangorp, Peter; Capiluppi, Andrea

Published in:
Journal of software-Evolution and process

DOI:
10.1002/smr.2267

Received: 23 August 2019 | Revised: 14 February 2020 | Accepted: 31 March 2020
DOI: 10.1002/smr.2267

WILEY Journal of Software: Evolution and Process

RESEARCH ARTICLE - EMPIRICAL

An empirical analysis of source code metrics and smart contract resource consumption

Nemitari Ajienka[1] | Peter Vangorp[2] | Andrea Capiluppi[3]

[1]Department of Computing and Technology, Nottingham Trent University, Nottingham, UK
[2]Department of Computer Science, Edge Hill University, Ormskirk, UK
[3]Department of Computer Science, University of Groningen, Groningen, The Netherlands

Correspondence
Nemitari Ajienka, Department of Computing and Technology, Nottingham Trent University, Nottingham, UK.
Email: nemitari.ajienka@ntu.ac.uk

Abstract

A smart contract (SC) is a programme stored in the Ethereum blockchain by a contract-creation transaction. SC developers deploy an instance of the SC and attempt to execute it in exchange for a fee, paid in Ethereum coins (Ether). If the computation needed for their execution turns out to be larger than the effort proposed by the developer (i.e., the *gasLimit*), their client instantiation will not be completed successfully.

In this paper, we examine SCs from 11 Ethereum blockchain-oriented software projects hosted on GitHub.com, and we evaluate the resources needed for their deployment (i.e., the *gasUsed*). For each of these contracts, we also extract a suite of object-oriented metrics, to evaluate their structural characteristics.

Our results show a statistically significant correlation between some of the object-oriented (OO) metrics and the resources consumed on the Ethereum blockchain network when deploying SCs. This result has a direct impact on how Ethereum developers engage with a SC: evaluating its structural characteristics, they will be able to produce a better estimate of the resources needed to deploy it. Other results show specific source code metrics to be prioritised based on application domains when the projects are clustered based on common themes.

KEYWORDS

abstract syntax-tree (AST), blockchain-oriented software (BOS), Chidamber and Kemerer (C&K), object-oriented (OO), object-oriented programming (OOP), smart contract (SC)

1 | INTRODUCTION

A blockchain is a shared ledger that stores transactions in a decentralised[1] peer-to-peer network of computers also known as *nodes*. Blockchain transactions can be composed of contract *creation* transactions and contract *function invoking* transactions. The former deploys and records a smart contract (SC) on the blockchain, and the latter causes the execution of a contract functionality.[2,3] The third transaction type is the token or cryptocurrency transfer transaction such as Bitcoin transfers on the Bitcoin Blockchain or Ether transfers on the Ethereum Blockchain. As a whole, the blockchain technology provides a decentralised, trustless platform that combines transparency, immutability and consensus properties to enable secure, pseudo-anonymous transactions.

SCs are the programmes stored in a blockchain by a contract-creation transaction. In the last few years, SCs have been used in different scenarios: in voting platforms to secure votes; to automatically process insurance claims according to agreed terms and postal companies for payments on delivery.[5]

Porru *et al*[6] defined the term *blockchain-oriented software* (BOS) as a software that contributes to the realization of a blockchain project. This definition includes both blockchain platforms (or networks), such as Bitcoin and Ethereum, and general blockchain software commonly referred to as decentralised apps (DApps).[7]

In a blockchain network, each node maintains a copy of the blockchain or ledger, and some nodes can also perform an activity known as mining. Miner nodes (or *miners*) have the responsibility of validating ledger transactions and appending new transactions sets (i.e., *block*) to the previous block, which then makes up a chain of blocks (i.e., *blockchain*). An SC is run on the blockchain by each miner deterministically replicating the execution of the SC bytecode on a local blockchain client. The miner that successfully appends the transaction in a proposed and approved block receives the transaction fee corresponding to the amount of computational resources (known as *gas*) that the execution has actually burned, multiplied by the unit fee, known as *gasPrice*.

In order to limit the amount of resources committed by a node to the contract execution, transactions have a *gasLimit* field to specify the maximum amount of gas that the sender is willing to pay. If an SC execution transaction requires more *gas* than the *gasLimit*, the execution terminates with an *out-of-gas* exception, and the blockchain state is rolled back to the initial state prior to the execution. In this case, the transaction sender pays all the *gasLimit* to the miner as a counter-measure against resource-exhausting attacks.[8]

In view of such attacks, researchers[9] have called for the need for a blockchain software engineering domain considering the impact of SC vulnerabilities or bugs[10] (e.g., Reentrancy and frozen ether[11,12,13]), poor programming practices[14] in the languages used to write the SC code (i.e., Solidity) and deterministic execution. Given the immutable nature of the ethereum blockchain, it is crucial to ensure that SCs are free from bugs and not vulnerable to attacks.[15] A recent example is the distributed autonomous organisation (DAO) SC hack that led to the loss of 3.6 million Ethers (equivalent to $761 million USD).

In this paper, we study whether the evaluation of the *gasLimit* can be informed by the structural characteristics of the SC itself, and whether the application domains of these contracts plays a role too. Specifically, we study if there is a correlation between the object-oriented metrics of an Ethereum blockchain SC and the amount of *gasUsed* to deploy it onto the blockchain. It is noteworthy that the focus of this paper is on the Ethereum blockchain that requires gas for SC deployment and invocation and not all blockchain platforms have an in-built cryptocurrency used to pay for transaction gas costs, for example, private or consortium blockchain platforms such as Hyperledger Fabric[16*†] and Corda[17,18‡].

The rationale for investigating source code metrics (and application domains) in relation to SC deployment costs also concerns the compilation of SCs into bytecode[§¶] before deployment. Before deployment, an SC needs to be encoded into ethereum virtual machine (EVM) friendly binary called bytecode, much like a compiled Java class[#] . Therefore to reduce deployment costs, developers need to modify the functionality of the SC in an understandable manner, that is, in source code format before the SC is converted to bytecode as there is no guarantee of the functionality of the SC after modifying the bytecode version.

The two null hypotheses that we will test in this work are as follows:

> $H_{1,0}$: there is no significant correlation between the object-oriented (OO) metrics of a SC and the *gasUsed* to deploy it
> $H_{2,0}$: the application domains of the SCs do not play a role in the correlations between OO metrics and *gasUsed*

The software engineering research community and practitioners alike have relied on the use of OO software metrics for evaluating design decisions, architecture quality and degradation of software. Metrics are useful to assess the internal quality of a software as well as the productivity of the development team.[19] "It is not possible to control what you do not measure; such statement is the basic wisdom on why we need to use metrics".[20]

Establishing a link between *gasUsed* and the underlying OO metrics could be beneficial for both the creators of the SC, and the users considering to invoke the contract off the blockchain. In both cases, an a priori correlation would help making a decision on the amount of *gas* needed to perform the executions and the resulting fee to be paid.

The above motivation is also shared by Porru et al,[6] which states "due to the distributed nature of the blockchain, specific metrics are required to measure complexity, communication capability, resource consumption (e.g., the so-called gas in the Ethereum system) and overall performance of BOS systems". Additionally, Ducasse et al[20] state that "due to the extremely fast growing pace of SC usage, in this new software paradigm measuring code quality is becoming as essential as in out-of-chain software development". In both cases, researchers emphasized the need for *gas* or resource consumption estimation and structural metrics extraction tools.[21] The following are the main contributions of our study:

- the adoption of OO metrics in the BOS engineering domain, and
- a novel empirical investigation of the link between OO software metrics and the resource (*gas*) required to deploy SCs on the Ethereum blockchain, to address the research question: *is there a significant relationship between static software metrics and the resource consumed when deploying SCs to the Ethereum blockchain?*

[*] Consensus implies that the participating nodes on the *decentralised*[1] blockchain network have to always agree on the state of the network. As such, consensus protocols such as the proof-of-work[4] are embedded in blockchain networks to ensure that each block in the chain is validated and participants are incentivised for validating transactions before new blocks are appended to the chain.

[†] Hyperledge Fabric SCs are written in GoLang.

[‡] Corda SCs are written in Kotlin.

[§] `Example bytecode: 0x6080604052348015610010576000080fd5b506040516020806102d...`

[¶] One byte is represented by two letters in the bytecode.

[#] The following steps usually need to occur prior to SC deployment: the SC is developed in a human-friendly programming language (such as Solidity); the program is then compiled into bytecode; the bytecode is included alongside other information in a contract creation transaction which is sent to the blockchain network for approval; once approved, a unique blockchain address for the SC is created and returned to the user or developer.

- a (publicly available) curated and manually verified data set‖ that maps the SCs from 11 Ethereum blockchain-oriented projects to their associated OO software metrics, and the supporting scripts to allow researchers to conduct further studies in this domain.

The rest of this paper is articulated as follows: Section 2 provides an overview of the OO metrics, Ethereum blockchain SCs and associated resource consumption. Section 3 describes the empirical approach that was used to extract the OO metrics, as well as the consumed resources. Section 4 summarizes the results, whereas Section 5 discusses the findings and provides further empirical insights. Section 6 discusses the threats to validity. Section 7 evaluates the related work, whereas Section 8 concludes.

2 | BACKGROUND

2.1 | Software structural and architectural metrics

Chidamber and Kemerer[22] recommended a suite of OO metrics**. It includes coupling between objects (CBO),[23] response for a class (RFC), weighted methods per class (WMC), depth of inheritance tree (DIT), number of children (NOC), and lack of cohesion in methods (LCOM). The purpose of these metrics is to provide a theoretical background for software measurements and complexity metrics.

The relevance of such metrics comes to prominence when there is the need to evaluate software quality, evaluate and enhance developer productivity, reduce maintenance resources and improve process.[24,25] For example, the C&K metrics have been adopted by researchers in various scenarios: when predicting software maintainability[26]; investigating class dependencies in OO software[27]; evaluating the impact of inheritance types on the metrics[28]; evaluating software cohesion and comprehension[29]; and as features in prediction models that predict failures and defects.[30-33] For example, CBO has been shown to be correlated to class quality (defect or error-proneness of a class).[23,34,35] In addition to the C&K metrics, Hegedűs investigated the nature of the typical structure of SCs in terms of their OO attributes with additional metrics[21] including source lines of code (SLOC), logical lines of code (LLOC), comment lines of code (CLOC), number of functions (NFs), McCabe's cyclomatic complexity[36] (McCC), nesting level (NL), nesting level without else-if (NLE), number of parameters (NUMPARs), number of statements (NOSs), number of ancestors (NOAs), number of attributes or states (NA) and number of outgoing invocations (NOIs) , that is, fan-out.

Establishing the importance of these metrics in this context, that is, identifying a significant link between the metrics and deployment costs of programmes deployed on the blockchain will be beneficial for especially novice SC developers in the blockchain industry still in its early days. At a higher level, such metrics will guide an inexperienced developer on areas of source code to modify or refactor in an attempt to keep deployment costs low.

At a much lower level, the gas or deployment costs are linked to each operation or bytecode, called Opcodes, which is understood and executed by the EVM,[37] which could be less understood by a novice developer with regards to refactoring. In some instances, it could cost around $3 USD to deploy one SC to the Ethereum blockchain††. Deploying a project composed of around 20 SCs ($60 USD) can be significant depending on the resources available to the project owner.

In addition to the C&K metrics,[22] this paper makes use of the metrics investigated by Hegedűs[21] (see the list below). We have also adopted the SolMet tool implemented in Java and provided in Hegedűs[21] for the parsing of the SCs and extraction of the OO metrics. In summary, the studied SC software metrics include the following:

- SLOC: source lines of code;
- LLOC: logical lines of code;
- CLOC: comment only lines of code;
- NF: number of functions;
- McCC: McCabe's cyclomatic complexity of the functions[38];
- NL: sum of the deepest nesting level of the control structures within functions[21];
- NLE: nesting level without else-if;
- NUMPAR: number of parameters per function;
- NOS: number of statements;
- NOA: number of ancestors;
- WMC: weighted methods per class;
- DIT: depth of inheritance tree;
- CBO: coupling between objects;
- NA: number of attributes or state variables); and lastly,
- NOI: number of outgoing invocations or functions called from a function in a SC.[21]

‖ The data set and associated tools used for the extraction of the metrics for this study are publicly available at: https://figshare.com/articles/Smart_Contract_Metrics_and_Deployment_Costs/10353731

** Generally referred to as Chidamber and Kemerer Java Metrics (CKJM) or C&K.

†† https://hackernoon.com/costs-of-a-real-world-ethereum-contract-2033511b3214

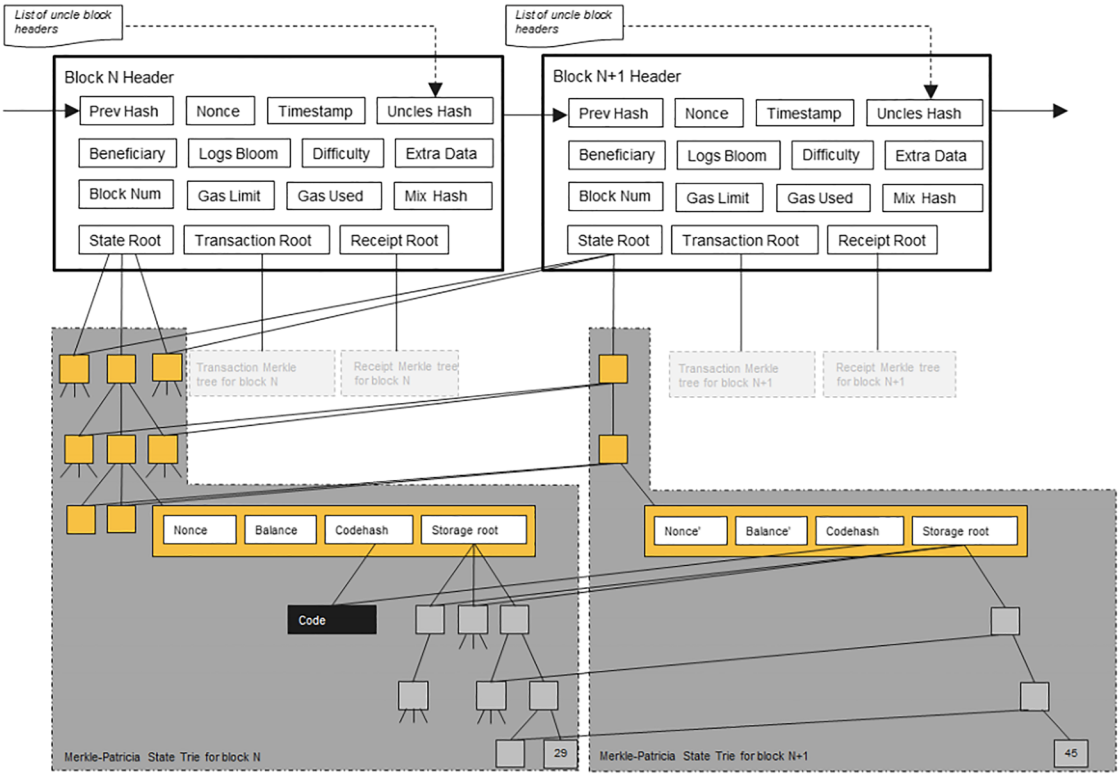


FIGURE 1 Blockchain and Ethereum architecture (adopted from Destefanis et al[39]). Each block of the chain consists of a set of transactions

2.2 | Ethereum blockchain and SCs

2.2.1 | Ethereum blockchain

A blockchain in summary is a shared ledger that stores transactions, composed of sets of information, in a decentralised peer-to-peer network of computers also known as nodes. Each node maintains a copy of the ledger, and some nodes can also perform an activity known as mining. Miner nodes (*miners*) have the responsibility of validating ledger transactions and appending new transactions sets (*block*) to the previous block, which then makes up a chain of blocks (*blockchain*). This data structure is what is referred to as a *blockchain*[‡‡], shown in Figure 1 (as adopted from Destefanis et al[39]). This figure also shows the components of each block including the resources consumed by its transaction components (in *gas* terms).

Miners use a predefined consensus protocol in order to agree on the validity of each block.[40] At any time, miners group their choice of incoming new transactions in a new block, which they intend to add to the blockchain. In most cases, the consensus protocol uses a probabilistic algorithm for electing the miner who will publish the next valid block in the blockchain. In the case of Ethereum, such a miner is the one who solves a computationally demanding cryptographic puzzle. This procedure is referred to as *proof-of-work*. All other nodes verify that the new block is correctly constructed (e.g., no virtual coin is spent twice) and update their local copy of the blockchain with the new block.

In the case of the Bitcoin blockchain platform, transactions are mostly based on the transfer of coins from one wallet (uniquely identified by an address) to another. On the other hand, Ethereum blockchain transactions can further be composed of (i) SC *creation* transactions and (ii) SC *function invoking* transactions. The former deploys and records a SC on the blockchain, and the latter causes the execution of a contract functionality. In this study, we are focusing on the former that is the deployment of a SC and its associated costs in relation to the structural attributes of the SC. The original white papers of the Bitcoin and Ethereum blockchains (Nakamoto and Bitcoin[2] and Buterin[3]) provide more in-depth details.

2.2.2 | Smart contracts

A SC is a programme stored in a blockchain by a contract-creation transaction. An SC is identified by a unique address[§§¶¶##] generated upon a successful creation transaction. An Ethereum SC address thus generally points to its executable code and a SC *state* consisting of (i) private storage, and (ii) the amount of virtual coins (Ether) it holds, that is, the contract balance.[39]

[‡‡] Transactions are grouped together into blocks, each hash-chained with the previous block.
[§§] Example SC address: `0x1A21f75140LK876351b8c0e9YBz1141fa3cB5b05`
[¶¶] Ethereum blockchain addresses are often represented as 40-character hexadecimal strings. These are usually saved with a hex prefix ("0x"), making them 42 characters long.
[##] The "0x" prefix means hexadecimal and it is a means by which programmes, contracts, and application program interfaces (APIs) understand that the input should be interpreted as a hexadecimal number. As an example, the (decimal) number 18 is "12" in hex. To remove any confusions with the number 12, adding 0x before 12 makes it clear that `0x12` is hexadecimal.

SCs and blockchain platforms have gained tremendous popularity in the past few years, and billions of US Dollars are currently exchanged through this technology. SCs can be applied to many different scenarios: they could be used in voting platforms to secure votes; insurance companies could use them to automatically process claims according to agreed terms programmed in the SC and postal companies for payments on delivery.[5]

Conceptually, Ethereum can be viewed as a huge transaction-based state machine, where its state is updated after every transaction and stored in the blockchain. The Ethereum blockchain users can transfer Ether coins from address to address or wallet to wallet using transactions, like in the case of Bitcoin. Additionally, they can invoke SC functionalities using contract invoking transactions.

One of the motivations for this study is the fact that SCs rely on a non-standard software life-cycle, according to which, for instance, delivered applications can hardly be updated, or bugs resolved by releasing a new version of the software. Since the release of the Frontier network of Ethereum in 2015, there have been many cases in which the execution of SCs managing Ether coins led to problems or conflicts.[13,41,42]

From a software development perspective, the SC code must satisfy constraints typical of the domain, such as (i) they must be light; (ii) their deployment on the blockchain must take into account the cost in terms of some crypto value; (iii) their operational cost also in terms of crypto value must be limited and (iv) they are immutable, since the bytecode is inserted into a blockchain block once and forever.[43]

The above constraints are due to the fact that SCs are self-enforcing agreements, that is, contracts implemented through a computer programme, whose execution enforces the terms of the contract. The long-term objective is to get rid of a central control authority, entity or organization that parties involved in a contract must trust, and delegate such role to the correct execution of a computer program instead. Such scheme can thus rely on a decentralised system automatically managed by machines.

The blockchain technology is the instrument for delivering the trust model conceptualized by SCs. Because SCs are stored on a blockchain, they are public[‖‖] and transparent, immutable and decentralised, and because blockchain resources are costly, their code size has to be taken into serious consideration. Immutability means that when an SC is created, it cannot be changed again.

2.2.3 | Implementing SCs

An SC's source code makes use of variables just like traditional imperative programmes. According to Dannen, "at the lowest level, the code of an Ethereum SC is stack-based bytecode, run by an EVM in each node. SC developers define contracts using high-level programming languages".[37] The widely adopted programming language for Ethereum blockhain SCs is Solidity, usually referred to by researchers and developers like Luu et al,[44] as "a JavaScript-like language which is compiled into EVM bytecode".

The EVM enables the Ethereum blockchain to be used as a platform for creating DApps. In addition, Solidity shares some OO programming concepts (e.g., classes and objects).[37,44]

The concept of a "class" (e.g., a Java class) in Solidity is realized through a "contract", which is a prototype of an object that lives on the blockchain. According to Zhang et al, a contract can be instantiated into a concrete decentralised application by a deployment transaction or a function call from another contract in the same way an object-oriented class can be instantiated into a concrete object at runtime.[45] At instantiation, a contract is allocated a distinct address[***] similar to a pointer in C/C++-like languages.[45]

As highlighted by Destefanis et al,[39] "once a SC is created at a blockchain address, it can then be invoked or called by sending a contract-invoking transaction to the address. A contract-invoking transaction typically includes the payment (in Ether) of the contract for its execution; and/or input data for a function invocation". A working example of this mechanism is described below.

2.2.4 | Resource consumption and *gas* system

An SC is run on the blockchain by each miner deterministically replicating the execution of the SC bytecode on a local blockchain client. This implies that in order to guarantee integrity across replications of the blockchain, the code must be executed in a strictly deterministic way[†††]. Solidity and in general high-level SC languages are Turing complete in Ethereum. Nevertheless, in a decentralised blockchain architecture Turing completeness may lead to certain issues. For example, the replicated execution of infinite loops may potentially freeze the blockchain network.

To ensure fair compensation for expended computation efforts across the network and limit the use of resources, miners in the Ethereum blockchain network are paid some fees, proportionally to the required computation. Specifically, each instruction in the Ethereum bytecode requires an amount of a resource referred to as *gas*, paid in Ether (the cryptocurrency used on the Ethereum blockchain). When developers or SC users send a contract-invoking transaction, they can specify the amount of *gas* they are willing to provide for the execution, called *gasLimit*,[46] as well as the price for each *gas* unit called *gasPrice*.

The miner that successfully appends the transaction in a proposed and approved block receives the transaction fee corresponding to the amount of *gas* that the execution has actually burned, multiplied by the *gasPrice*. If an SC execution requires more *gas* than the *gasLimit*, the execution terminates with an *out-of-gas* exception, and the blockchain state is rolled back to the initial state prior to the execution. In this case, the user pays the whole *gasLimit* to the miner as a counter-measure against resource-exhausting attacks.[8] Hence, the rationale for the ability to

[‖‖] It is noteworthy that there are also private versions of the Ethereum blockchain. However, we are focusing on the public Ethereum blockchain network.
[***] Example SC Address: `0x425372c6ac9d559a197a08a3854e0ddea1a00d2c`
[†††] For instance, the generation of random numbers may be problematic

```
1  pragma solidity ^0.4.17;
2  contract Course {
3      address private moduleLeader; // smart contract owner
4      mapping (address => bool) private students;
5
6      modifier isModuleLeader() {
7          require(msg.sender == moduleLeader);
8          _; // the rest of a function can be executed after above condition is
               met.
9      }
10
11     constructor() public {
12         moduleLeader = msg.sender;
13     }
14
15     function addStudent(address id, bool include) isModuleLeader public
16     {
17         students[id] = include;
18     }
19
20     function getStudentStatus(address id) public view returns (bool) {
21         return students[id];
22     }
23 }
```

FIGURE 2 Smart contract example

```
1  pragma solidity ^0.4.17;
2  import "./DataStorage.sol";
3  contract Logic {
4      DataStorage dataStorage;
5
6      constructor(address _address) public {
7          dataStorage = DataStorage(_address);
8      }
9      function f() public {
10         bytes32 key = keccak256("emergency");
11         dataStorage.setUintValue(key, 911);
12         dataStorage.getUintValue(key);
13     }
14 }
```

FIGURE 3 Logic.sol smart contract importing and using functionalities of DataStorage.sol smart contract

estimate in advance the amount of *gas* required for a contract deployment or invoking transaction and to refactor the SC due to the availability of *gas* resources prior to deployment.

2.2.5 | Working example

Figure 2 depicts a basic example of a University Course SC. The SC stores the unique blockchain ID of students and permits only the module leader of the course to add and change the status of students. A contract-creation transaction containing the EVM bytecode for the SC in Figure 2 is sent to miner nodes in the Ethereum blockchain network. Eventually, the transaction will be accepted in a block, and all miners will update their local copy of the blockchain: first, a unique address for the contract is generated in the block, then each miner locally executes the constructor (Line 11) of the Course contract, and a local storage is allocated in the blockchain. Finally the EVM bytecode of the SC is added to the storage.

When a contract-invoking transaction is sent to the unique address of the Course SC to interact with a function, all information about the invoke message sender or the blockchain address from which the function is called, the amount of Ether sent to the contract, and the input data of the invoking transaction are stored in a default variable called `msg`.

When the `addStudent()` function (Line 15) is invoked, a transaction is sent to the SC on the blockchain. However, the function execution only begins after the condition in the modifier (Line 6) is successfully met. The condition in this example specifies that only the SC owner (i.e., the user who created or deployed the contract to the blockchain by calling the constructor) can add a new student by invoking the `addStudent()` (Line 15) function. Without the modifier `isModuleLeader` appended to the function declaration, anyone would be able to interact with this function. The `getStudentStatus()` (Line 20) function does not have this modifier because anyone is permitted to call this function or interact with this function (module leader or student) to check the enrollment status of a student.

To demonstrate an example of the link between the size metrics and the *gasUsed* metric, the *gasUsed* consumed when the SC in Figure 2 is deployed is 226,805. However, adding more lines of code to import and make use of the functionality in a library or SC called *SafeMath.sol* (e.g., `studentCount = SafeMath.safeAdd(studentCount, 1);`) increases the *gasUsed* to 259,257 (Figure 3).

Project	GitHub Repository https://github.com/	# SCs	# Contributors
Airbloc token	airbloc/token	4	3
Decentralised microinsurance	Denton24646/LDelay	2	2
DEXY token exchange	DexyProject/protocol	2	5
Gnosis prediction market	gnosis/pm-contracts	22	10
Grapevine World token and crowdsale	GrapevineWorld/crowdsale-contracts	4	2
Kleros	kleros/kleros	1	14
Monerium	monerium/smart-contracts	15	2
Realitio (crowd-sourced SC verification)	realitio/realitio-contracts	2	2
Synthetix	Synthetixio/synthetix	3	12
Token-curated registry	kangarang/is-tcr	5	11
TrueUSD token	trusttoken/trueUSD	6	4

TABLE 1 Selected Ethereum blockchain-oriented software sample

3 | METHODOLOGY

3.1 | Study sample

Kalliamvakou et al, investigated the quality and properties of data available from GitHub[47] and identified various potential perils to be considered when mining GitHub as a source of data on software development. Based on their study, we adopted the following search criteria when selecting case studies of BOS:

- The repository should be an Ethereum BOS project (with Solidity as the main language) and not a library or tutorial.
- The project should have a significant number of commits. A minimum of between 5 to 10 commits. Similar criterion has been adopted in prior work[48,49] to guarantee that we only analyse projects where there is some development activity.
- It should not be a personal project: it should have at least two active contributors. Similar filtering criterion is used in prior work.[50]
- To exclude inactive projects, the projects must have at least one commit in the last 12 months preceding the data collection.[51]

Based on the aforementioned case study selection criteria, the chosen case studies are listed in Table 1 including the number of deployed and studied SCs and contributors per project.

Using the GitHub Search API[‡‡‡], we searched repositories using the selection criteria described above. First, we used a simple `curl` command to download details of all projects with Solidity as the main language and sorted by the number of stars in descending order to enable us to identify the most successful Solidity projects hosted on GitHub as case studies. This gave us 1,179 projects in total. The "success" of the projects is determined by the number of *stars* received by the community of GitHub users and developers, as a sign of appreciation. We used this approach to stratified sampling because the projects obtained by this filter are likely to be used by a large pool of users,[52] and active in terms of the number of commits[47,53] in the last 3 months preceding the sample collection for the study. Prior studies have also adopted similar selection criteria[54,55] when analysing software repositories hosted on GitHub.

We further narrowed the sample down to 266 repositories that contain a *Truffle* project (Truffle[§§§] is a framework or collection of command-line tools for developing, testing, deploying and managing Solidity SCs and their dependencies) by using the GitHub Search API to extract the projects that contained the term "truffle" in their `README.md` file[¶¶¶].

After that, the GitHub Search API output consisting of information relating to the projects was parsed using a simple shell script to get the `clone_url` and clone the source of each project from GitHub.

We then inspected the number of contributors and activity and discarded those projects that did not compile (for deployment) or meet the selection criteria listed above (e.g., projects that have been inactive in the current year or have only one contributor). This was labour-intensive and a similar criterion has been adopted in a related study on SC metrics by Vandenbogaerde [56###] and helps to ensure that the same standard applies to all studied projects reducing the chance of compilation issues. In addition, tools from the truffle framework have been used in the later parts of the methodology to interact with and deploy the SCs in order to extract the deployment costs. The final sample consists of 11 projects composed of 66 deployed SCs[‖‖‖]. Similarly, 11 projects written in C/C++ were studied in Norick et al[57] given constraints such as the lack of consistency in stored information from one project to another and challenges in accessing the source code repository for a project.

The source code of the final sample of projects including the SC source code is used in the following parts of the methodology to extract the required metrics for the study.

[‡‡‡] https://developer.github.com/v3/search/
[§§§] https://truffleframework.com/
[¶¶¶] The GitHub Search API states that requests that return multiple items will be paginated to 30 items by default. Therefore, we have used pagination to specify further pages with the ?page parameter as well as set a custom page size up to 100 with the ?per_page parameter. This meant we had to run the command three times for the 266 projects (≃ 3 pages).
[###] We ended up with the following query/command: `curl` https://api.github.com/search/repositories?q=truffle+in:readme+language:solidity&sort=stars&order=desc&page=1&per_page=100
[‖‖‖] The list of projects and all the extracted metrics for this study are publicly available at https://figshare.com/articles/Smart_Contract_Metrics_and_Deployment_Costs/10353731

```
// File: contracts/DataStorage.sol
pragma solidity ^0.4.17;
contract DataStorage {

    .......

}

// File: contracts/Logic.sol
pragma solidity ^0.4.17;

contract Logic {
    DataStorage dataStorage;

    .......

}
```

FIGURE 4 Flattened Logic.sol smart contract with the previously imported dependency (DataStorage.sol smart contract) combined in one *flat* Solidity file

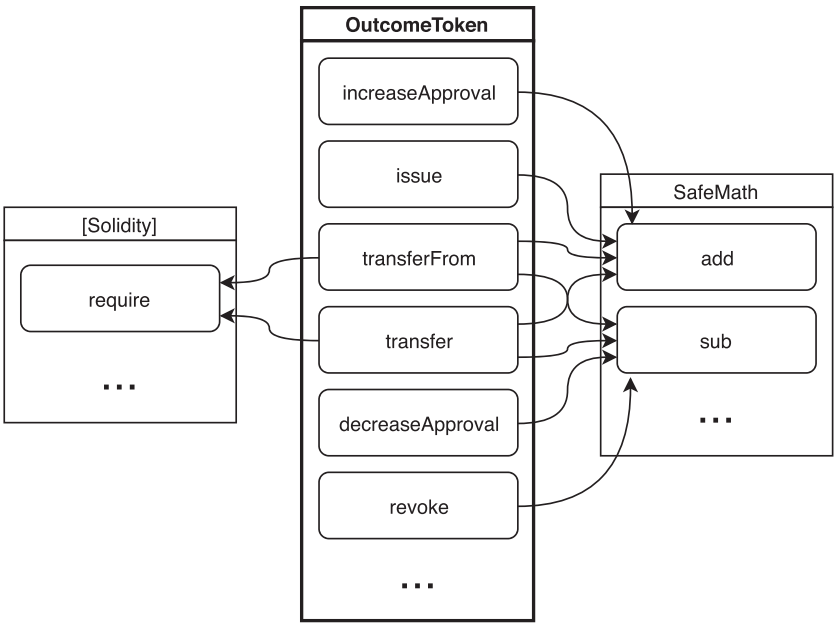


FIGURE 5 Example call graph extracted from the Gnosis *OutcomeToken.sol* smart contract

3.2 | Extracting the OO software metrics

The OO metrics were extracted using a tool called SolMet, provided and used in Hegedűs.[21] However, in order for the metrics to be extracted the SCs had to be flattened: in other words all the dependencies, that is, imported SCs and libraries, had to be combined with the dependent SC into one Solidity *.sol* file. This step was labour-intensive and required that all broken imports had to be manually resolved in order for source code dependencies to be found. This step is also required for the verification of publicly used SC source code on Etherscan[****], a process that enables transparency and trust in the source code of publicly used SCs. For this study, the flattening was performed using the *truffle-flattener* tool[††††].

As an example, Figure 4 shows a `Logic.sol` SC that utilises the functionalities of a `DataStorage.sol` SC with the source code of both contracts in one file.

Once the SCs were flattened, they were then parsed using SolMet to perform the extraction of the structural and architectural metrics.[5859] We could also verify some of the coupling metrics (e.g., RFC and LCOM) by extracting the call graph (Figure 5) and data dependencies from each contract using the Slither static analysis tool[‡‡‡‡]. The source code was also inspected and cross-checked against the extracted metrics to mitigate any errors.

3.3 | Extracting the consumed resources (i.e., *gasUsed*)

Deploying the SCs to the Ethereum blockchain network and deriving the resources consumed in terms of *gas* costs requires a test Ethereum blockchain network node to be set up as well as the availability of some test resources or the Ether crypto currency to pay the mining costs. To

[****] Etherscan (https://etherscan.io/) allows users to explore and search the Ethereum blockchain for transactions, addresses, tokens, prices and other activities taking place on Ethereum.
[††††] https://github.com/nomiclabs/truffle-flattener
[‡‡‡‡] https://github.com/trailofbits/slither

avoid this bottleneck, we have used the Ganache command line tool[§§§§] that is one of the tools in the suite of tools for Ethereum SC development provided by the Truffle community[¶¶¶¶]. The tool enables rapid development and testing of SCs with a better network latency compared with waiting for transactions to be mined by a miner node and appended to the live blockchain network. It simulates a full Ethereum blockchain and client behavior and provides free Ether and accounts with which to perform SC tests. The tool can be installed and used on a local machine. An online web-based variant of this tool is also available called Remix. As described by the authors of the GitHub project[####], "Remix is a browser-based compiler" and integrated development environment that enables users to build Ethereum SCs with the Solidity programming language and to debug transactions[‖‖‖‖]. Remix also enable the testing of SCs via unit tests written using tape[*****]. However, usage of Remix relies on internet connection.

Once a SC has been deployed to the blockchain using Truffle, the `getTransaction(hash)` Ethereum function[†††††] provided by the *web3.js* JavaScript library [‡‡‡‡‡] can be used to get details of a SC deployment or method call transaction sent to the blockchain including the *gasPrice* paid to the miner node that added the transaction to a block appended to the blockchain, whereas `getTransactionReceipt(hash)` provides the transaction receipt that includes the actual *gasUsed* on the blockchain. The *gasCost* is then calculated as the product of the *gasPrice* and *gasUsed* by the transaction. For each analysed SC, we have written a tool in JavaScript, which uses the *web3.js* library to extract these resource metrics upon deployment.

3.4 | Statistical test—Spearman's correlation

This section describes the computation of statistical tests in order to answer the research question: *is there a significant relationship between static software metrics and the resource consumed when deploying SCs to the Ethereum blockchain?* The relationship under investigation is the relationship between the extracted OO metrics and the *gasUsed* during the deployment of each SC outlined in Section 3.1.

Given the BOS project described in Section 3.1, for each metric we created two vectors, one with the values of the metric (e.g., CBO) and the other with the *gasUsed* during deployment. The null hypothesisH_0 to be tested is as follows:

- H_0: there is no significant correlation between the OO metrics of a SC and the *gasUsed* to deploy it

The correlation between the two vectors is evaluated using the Spearman's rank correlation coefficient[60] in `R`, for example, `result <- cor.test(SLOC, gasUsed and method="spearman")`. Various other correlation coefficients have been considered including Pearson and Kendall. However, for Pearson's to be valid, the data have to follow a normal distribution.[60,61] Spearman's rank correlation, a non-parametric test, was chosen because the results of a Shapiro–Wilk normality test on the OO metrics, and the *gasUsed* revealed that the data do not follow a normal distribution. Kendall'sτ would have been used in smaller sample sizes and where there are multiple values with the same score[62] for all the metrics under investigation.

We reject the null hypothesis at the 99% confidence level. In other words, if the rank correlation coefficient proves to be statistically significant at the$\alpha < 0.01$ level, we will reject the null hypothesis and fail to reject the alternative hypothesis$H_{1,1}$: there is a significant correlation between the OO metrics of a SC and the *gasUsed* to deploy it. The results derived for all projects are presented in Section 4.

4 | RESULTS AND DISCUSSION

This section presents and discusses the empirical results of this study in detail. As described in Section 3.4, we have evaluated the correlation between each OO metric and the *gasUsed* using the Spearman's rank correlation method. The value of the correlation coefficientρ lies in the range$[-1; 1]$, where-1 indicates a strong negative correlation and 1 indicates a strong positive correlation. We adapt the categorisation for correlation coefficients in Marcus and Poshyvanyk[63] ($[0 - 0.1]$ *insignificant*,$[0.1 - 0.3]$ *low*,$[0.3 - 0.5]$ *moderate*,$[0.5 - 0.7]$ *large*,$[0.7 - 0.9]$ *very large*, and$[0.9 - 1]$ *almost perfect*) if theρ coefficient proves to be statistically significant at the$\alpha = 0.01$ level.

We present and discuss below the results for the GitHub project with the most SCs (i.e., the Gnosis project); then, we evaluate the results for the overall set of projects studied to answer the research question: *is there a significant relationship between static software metrics and the resource consumed when deploying SCs to the Ethereum blockchain?* The impact of the results for researchers and practitioners is also discussed.

4.1 | Spearman's correlations—Gnosis project

In this section, we show the results of the correlation analysis for the project with the largest number of SCs of our sample (the Gnosis project). Tables 2 and 3 show the raw data for the metrics gathered, together with the evaluation of the *gasUsed* attribute, per SC. We split these data into

[§§§§] https://github.com/trufflesuite/ganache-cli
[¶¶¶¶] https://truffleframework.com/
[####] https://github.com/ethereum/remix-ide
[‖‖‖‖] The integrated development environment (IDE) can be found at: https://remix.ethereum.org
[*****] https://www.npmjs.com/package/tape
[†††††] A transaction hash is an identifier used to uniquely identify a particular transaction in the blockchain.
[‡‡‡‡‡] https://github.com/ethereum/wiki/wiki/JavaScript-API#web3ethgettransaction

TABLE 2 Chidamber and Kemerer (C&K) metrics for the *Gnosis* project and Spearman's rank correlation versus *gasUsed* (post-deployment)

Smart contract	WMC	DIT	NOC	CBO	SLOC	RFC	LCOM	gasUsed
Campaign	5	1	0	1	64	30	0	1,971,730
CampaignFactory	2	0	0	1	24	2	1	923,821
CategoricalEvent	10	2	0	1	19	27	53	1,381,002
CentralizedOracle	6	1	0	0	33	6	6	470,403
CentralizedOracleFactory	2	0	0	1	16	2	1	697,528
DifficultyOracleFactory	1	0	0	1	10	1	0	316,405
EventFactory	3	0	0	3	62	9	1	2,313,772
FutarchyOracle	7	1	0	1	61	28	6	1,715,623
FutarchyOracleFactory	2	0	0	3	69	3	1	1,246,926
LMSRMarketMaker	11	1	0	1	116	49	1	1,644,921
MajorityOracle	5	1	0	0	51	7	3	471,759
MajorityOracleFactory	2	0	0	1	16	2	1	570,570
OutcomeToken	15	1	0	0	26	30	45	1,468,848
ScalarEvent	10	2	0	1	32	26	26	1,680,640
SignedMessageOracle	6	1	0	0	36	12	2	622,976
SignedMessageOracleFactory	2	0	0	1	17	3	1	608,857
StandardMarket	17	2	1	1	148	54	35	3,594,149
StandardMarketFactory	2	0	0	3	14	2	1	917,649
StandardMarketWithPriceLogger	25	3	0	1	62	49	72	3,855,961
StandardMarketWithPriceLoggerFactory	2	0	0	1	17	2	1	1,103,518
UltimateOracle	11	1	0	1	87	33	10	1,295,451
UltimateOracleFactory	2	0	0	2	49	2	1	863,412
Spearman's rank correlation ρ	0.65	0.52	0.33	0.28	0.62	0.74	0.38	
p value	<0.01	0.01	0.14	0.20	<0.01	<0.01	0.08	

Abbreviations: CBO, coupling between objects; DIT, depth of inheritance tree; LCOM, lack of cohesion in method; NOC, number of children; RFC, response for class; SLOC, source lines of code; WMC, weighted methods per class.

two tables for easier reference and visualisation. Considering the Spearman's correlation coefficients, we obtain a *very large* correlation between the RFC attribute and the *gasUsed*, and several *large* correlations between other metrics: WMC and DIT among the C&K metrics, but also SLOC, LLOC, CLOC, NF, NL NLE, NUMPAR, NOS and NOI all show a ρ larger than 0.5 in the correlation with the *gasUsed* measurement.

These results demonstrate that for the SCs in the Gnosis project, the *gasUsed* attribute is more sensitive to the size measurements (SLOC, LLOC but also WMC and RFC) and less to the structural characteristics (CBO, NOC or LCOM). Observing the values of the structural attributes in Table 2, the analysed SCs are structurally simple OO classes, as reflected by the DIT (which also shows a moderate correlation with *gasUsed*), LCOM, NOC and CBO values. In the Gnosis project, the *gasUsed* shows a remarkable correlation with the size attributes (e.g., SLOC, NL and NOS).

These strong correlations are mirrored by the correlations that we observed between various OO attributes, as displayed in the correlation matrix of Figures 6 (the size of the circles is proportional to the strength of the correlation coefficients). The insignificant correlations (e.g., correlation< 0.01) are crossed out for clarity.

When the OO attributes possess a large or very large correlation between each other, a corresponding large correlation with *gasUsed* are to be expected. The large correlations with *gasUsed* are also expected given the bias and statistical power of the sample size (a single project), and a relationship may appear even though none exists.[64]

4.2 | Spearman's correlations—overall sample

The same approach used for the single Gnosis project was applied to all the data in the sample. Table 4 shows the rank correlations between each attribute and the *gasUsed* established earlier. We group metrics for which we obtained *moderate* levels of correlation, and the metrics for which we found *large* coefficients.

Similarly, to the Gnosis project, the overall sample of projects studied shows statistically significant (p value < 0.01) and moderate ($\rho = 0.5$) correlation between the *gasUsed* metric and the DIT metric. In contrast to the Gnosis project, the overall sample of projects studied shows statistically significant (p value < 0.01) and moderate ($\rho = 0.5$) correlations between the *gasUsed* metric and the following metrics: NOS, NOI and NOA. On the other hand, we observed low ($\rho = 0.3$ or 0.4) but statistically significant correlations between the *gasUsed* metric and the following metrics: SLOC, NF, WMC, NA and Average NOI. For these metrics, we can reject the null hypothesis but fail to reject the alternative hypothesis ($H_{1,1}$: there is a significant correlation between the OO metrics of a SC and the *gasUsed* to deploy it).

For the other metrics with insignificant correlation (p value > 0.01) such as the LLOC, CLOC, NL, NLE, NUMPAR, CBO, Avg. McCC, Avg. NL, Avg. NLE, Avg. NUMPAR and Avg. NOS, we cannot reject the null hypothesis. Figures 7a to 8b show scatter plots for the source code metrics highlighted in Table 4 that share the strongest and statistically significant correlations with the *gasUsed* metric.

TABLE 3 Additional objective-oriented (OO) metrics and Spearman's rank correlation versus *gasUsed* (post-deployment) for the Gnosis project

Smart contract	LLOC	CLOC	NF	NL	NLE	NUMPAR	NOS	NOA	NA	NOI	gasUsed
Campaign	64	17	5	1	1	1	30	2	0	17	1,971,730
CampaignFactory	24	17	1	0	0	6	3	0	1	1	923,821
CategoricalEvent	19	11	2	0	0	0	6	2	0	5	1,381,002
CentralizedOracle	33	13	4	0	0	2	9	3	0	2	470,403
CentralizedOracleFactory	16	12	1	0	0	1	3	0	1	1	697,528
DifficultyOracleFactory	10	9	1	0	0	1	2	0	0	1	316,405
EventFactory	62	24	2	0	0	7	13	0	5	6	2,313,772
FutarchyOracle	61	19	5	3	3	1	28	3	0	14	1,715,623
FutarchyOracleFactory	69	23	1	0	0	9	6	0	3	1	1,246,926
LMSRMarketMaker	115	82	7	6	6	20	63	1	2	41	1,644,921
MajorityOracle	52	11	3	5	4	0	27	3	0	6	471,759
MajorityOracleFactory	16	12	1	0	0	1	3	0	1	1	570,570
OutcomeToken	26	19	2	0	0	4	8	2	1	4	1,468,848
ScalarEvent	32	14	2	2	1	0	17	3	0	9	1,680,640
SignedMessageOracle	36	20	4	0	0	9	10	3	0	2	622,976
SignedMessageOracleFactory	17	15	1	0	0	4	4	0	1	2	608,857
StandardMarket	148	46	9	6	6	16	73	3	0	35	3,594,149
StandardMarketFactory	14	14	1	0	0	3	3	0	1	1	917,649
StandardMarketWithPriceLogger	62	34	8	2	2	11	21	2	0	14	3,855,961
StandardMarketWithPriceLoggerFactory	17	15	1	0	0	4	3	0	1	1	1,103,518
UltimateOracle	97	26	9	2	2	3	37	3	0	16	1,295,451
UltimateOracleFactory	49	17	1	0	0	6	3	0	1	1	863,412
Spearman's rank correlationρ	0.62	0.68	0.56	0.51	0.51	0.33	0.62	0.25	-0.02	0.68	
p value	<0.01	<0.01	<0.01	0.02	0.02	0.13	<0.01	0.25	0.9	<0.01	

Abbreviations: CLOC, comment lines of code; LLOC, logical lines of code; NA, number of attributes or states; NF, number of fumctions; NL, nesting level; NLE, nesting level without else-if; NOA, number of ancestors; NOI, number of outgoing invocation; NOS, number of statement; NUMPAR, number of parameter.

In Section 5, we further discuss the impact and potential applications of our empirical findings as well as provide an empirical investigation into the causal relationship between the source code metrics and the *gasUsed* metric by analysing their association with the bytecode size of SCs using the example SC in Figure 3 as a case study.

5 | DISCUSSION

In this section, we discuss the impact of the empirical results outlined in Section 4.2 laying emphasis on the moderately correlated metrics in Section 5.1. Furthermore, in Section 5.3, based on the notion that correlation does not imply causation,[64] we empirically investigate the causal relationship between the *gasUsed* metric and the moderately correlated source code metrics based on their association with the bytecode of the SCs using a case study.

In practice, the results demonstrate based on the studied sample that the inheritance based metrics NOA and DIT, the NOS size metric and the structural NOI metric are good indicators of the *gasUsed* metric when looking at the overall sample and can be used to guide practitioners when carrying out refactoring[65,66] to manage *gas* costs based on available resources. These results can also guide SC developers in the selection of which SCs they can engage with, and the amount of *gas* that they will be expected to spend on the deployment transaction, because the metrics show some strong correlations with the *gas* effectively used.

5.1 | Correlation between OO metrics and *gasUsed*

Considering the overall sample of blockchain-oriented projects studied, the OO metrics observed as having the highest correlations with the *gasUsed* metric are the NOS, DIT, NOA and NOI.

5.1.1 | Number of statements

In summary, in computer programming, a statement is a command or instruction given to the computer to perform. In most programming languages, statements are ended with a semi-colon to distinguish between different sets of instructions. Statements can be composed of internal components (i.e., expressions that are a combination of one or more constants, variables, operators and functions that the programming language interprets).

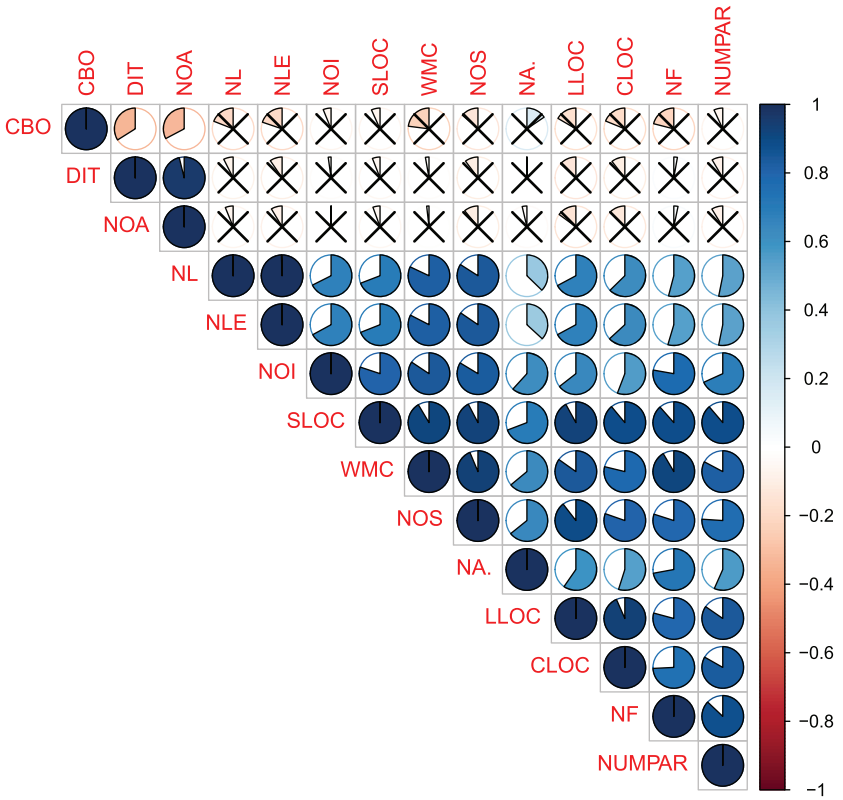


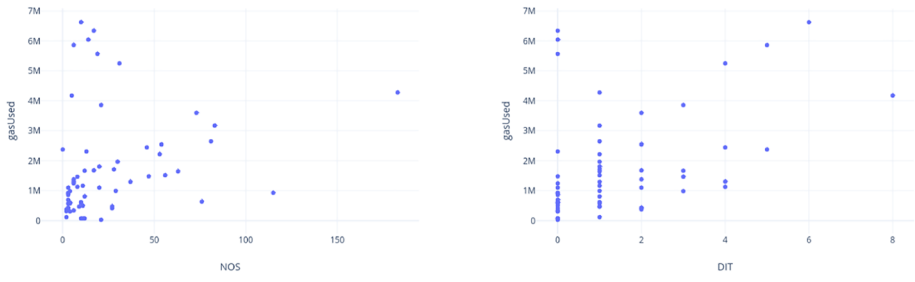
FIGURE 6 Correlation matrix for the source code metrics of the sampled contracts (insignificant correlations [i.e.,< 0.01] are crossed out). CBO, coupling between objects; CLOC, comment lines of code; DIT, depth of inheritance tree; LCOM, lack of cohesion in method; LLOC, logical lines of code; McCC, McCabeŠs cyclomatic complexity; NA, number of attributes or states; NF, number of fumctions; NL, nesting level; NLE, nesting level without else-if; NOA, number of ancestors; NOC, number of children; NOD, number of dependencies; NOI, number of outgoing invocation; NOS, number of statement; NUMPAR, number of parameter; OO, objective-oriented; SLOC, source lines of code; WMC, weighted methods per class

TABLE 4 Spearman's rank correlation results for source code metrics versus *gasUsed* metric (post-deployment)

OO metric	Spearman's ρ	*p* value
SLOC	0.4	0.0005
LLOC	0.1	0.5
CLOC	0.1	0.5
NF	0.3	0.01
WMC	0.3	0.01
NL	0.2	0.1
NLE	0.2	0.1
NUMPAR	0.2	0.2
NOS	0.5	0.0002
DIT	0.5	0.0002
NOA	0.5	0.0001
NOD	NA	NA
CBO	0.3	0.05
NA	0.4	0.0002
NOI	0.5	0.0001
Avg. McCC	0.10	0.4
Avg. NL	0.10	0.4
Avg. NLE	0.10	0.4
Avg. NUMPAR	-0.2	0.2
Avg. NOS	0.3	0.05
Avg. NOI	0.3	0.01

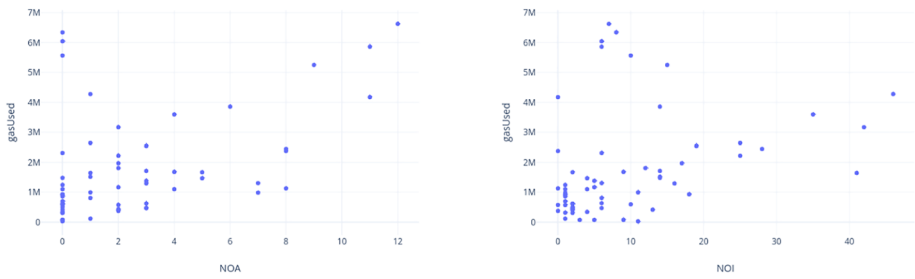
Abbreviations: CBO, coupling between objects; CLOC, comment lines of code; DIT, depth of inheritance tree; LCOM, lack of cohesion in method; LLOC, logical lines of code; McCC, McCabeŠs cyclomatic complexity; NA, number of attributes or states; NF, number of fumctions; NL, nesting level; NLE, nesting level without else-if; NOA, number of ancestors; NOC, number of children; NOD, number of dependencies; NOI, number of outgoing invocation; NOS, number of statement; NUMPAR, number of parameter; OO, objective-oriented; SLOC, source lines of code; WMC, weighted methods per class.

Our empirical results have shown that the number of statements or instructions in a SC can be a useful indicator of the required deployment costs of the SC. Essentially, the NOS metric is a size metric derived by counting the number of statements there are in a computer programme, which in this case is a SC. Specifically, in our studied sample of blockchain-oriented projects the NOS metric showed a significant moderate (ρ = 0.5) correlation with the *gasUsed* metric. This implies a strong relationship between the number of statements and the *gasUsed*.



(A) gasUsed vs NOS (B) gasUsed vs DIT

FIGURE 7 Spearman's rank correlation plots for source code metrics ((A) gasUsed vs. NOS and DIT) that show the strongest and statistically significant correlations with the *gasUsed* metric (post-deployment). DIT, depth of inheritance tree; NOS, number of statement



(A) gasUsed vs NOA (B) gasUsed vs NOI

FIGURE 8 Spearman's rank correlation plots for source code metrics ((A)gasUsed vs. NOA and (B) gasUsed vs. NOI) that show the strongest and statistically significant correlations with the *gasUsed* metric (post-deployment). NOA, number of ancestor; NOI, number of outgoing invocation

Comparison with traditional OO programming

It is traditionally expected that the SLOC metric will large correlation relationship with the *gasUsed* metric. However, our results show a stronger relationship with the NOS metric that is a component of the SLOC metric. This result is interesting and very distinct with practical applications as a weaker correlation strength is observed with the SLOC metric. This means that not all the source lines of are important when considering the *gasUsed* metric and not all lines of code affect the *gasUsed* for deployment but only statements specifically.

For practitioners

This result has actionable insights in practice for practitioners as it specifically pinpoints the lines of code that need more attention and practitioners will be able to optimise deployment resources by minimising the NOS of their SCs.

5.1.2 | DIT and NOA

The NOA metric is a count of the number of ancestors a SC inherits functionality from. Traditionally, in the OO software domain, NOA has been defined as the number of superclasses (both directly and indirectly inherited) of a class.[67] On the other hand, DIT is a measure of the location of a class in the inheritance hierarchy. Our empirical results have shown the *gasUsed* metric is moderately correlated ($\rho = 0.5$ and p value <= 0.01) with both DIT and NOA inheritance-based metrics.

Comparison with traditional OO programming

In traditional OO programming, researchers have identified a link between DIT and maintenance efforts. The deeper a Java class is in the inheritance hierarchy, the higher the total number of methods it is likely to inherit[22] making the behaviour of the class less predictable. Khalid et al, state that "DIT is directly proportional to complexity" (i.e., an increased DIT will lead to higher maintenance efforts),[68] which means that deeper trees lead to a higher design complexity since more methods and classes are involved.

In this study, the DIT metric also measures the position of an SC in the inheritance hierarchy (taking into consideration the deepest hierarchy). Interestingly, in relation to *gasUsed*, the DIT metric shows a significantly moderate correlation. This implies that the more methods or functionality an SC inherits, the more resources are required for its deployment to the ethereum blockchain network.

Differently from the DIT metric that computes the position of the SC in the deepest hierarchy, the NOA metric counts all ancestors from which an SC inherits from. In relation to DIT, the NOA metric has also been found to have a link to complexity and increased maintenance needs. As such, the NOA metric has been proposed as an alternative to the DIT metric in traditional OO programming given that the theoretical viewpoints of both metrics are similar and the NOA metric captures the environments from which the class inherits. The DIT and NOA metrics for fault-prone classes has also found to be higher and overlapping[69] in prior studies. Showing their interchangeability when measuring software complexity and fault-proneness.

Similarly, our empirical results have shown a moderate positive correlation between the NOA metric (as well as DIT) and the *gasUsed* metric in the SC programming domain. This shows that an increase in NOA (as well as an increase in DIT) can lead to an increase in the deployment costs

(gas) required for SC deployment. Thus, optimising the NOA of an SC will further minimise the required deployment resources. Furthermore, similar to the traditional OO software domain, the NOA and DIT metric can be used interchangeably in the SC domain as we have observed the same level of correlation (0.5) in our overall sample of blockchain-oriented projects. Researchers can further investigate the

In a study on fault prediction of OO software classes, both inheritance-based metrics DIT and NOC affected the potential of faults within a class because: deeper trees constitute greater design complexity because there are more methods a class can inherit. If there are greater number of DIT, it is difficult to predict the class behavior. In addition, the greater number of children, the greater the possibility of improper abstraction of the parent class.[70] A feasible topic in the SC domain will be an investigation of DIT and NOA for SC bug prediction and whether both metrics can be used interchangeably in this scenario.

For practitioners

From another point of view, the presence of a moderate significant correlation with inheritance based metrics DIT and NOA but not CBO or SLOC, implies in practice that inheritance can be reduced to reduce *gas* costs while utilising CBO to add to the functionality of a SC. This can be done by utilising the functionalities in already existing and deployed SCs or libraries to minimise deployment costs as opposed to inheriting functionality or importing large contract code into a base contract before deployment. As this will lead to high deployment costs each time there is a need to maintain the SC. Notwithstanding, attention is to be paid to the average fan-out of all functions in a SC. In traditional software development, studies have shown that high CBO reduces software quality; however, statistically, in the SC domain, a high CBO provides a useful option for maintenance.

Our results also provide a statistical backing for the contract decorator design pattern proposed by Liu et al,[71] and the external or segregated storage design pattern[§§§§§72] for SCs in view of deployment costs. The external storage pattern supports the storage of SC data in a different SC (making use of CBO) to give practitioners the flexibility to switch to a different SC with newly implemented functionality while retaining storage in another deployed contract. This will cost less *gas* if the SC has to be updated and redeployed and all the data stored in the old version is to be migrated into the new version in turns.

Another design pattern that utilises CBO but supports maintainability is the Satellite pattern.[41,72] It solves the problem of deploying a new contract instance when there is need to update its functionality. This is achieved through the creation of distinct satellite SCs that contain certain contract functionality. The addresses of the satellite contracts are then stored in a base contract that calls or makes reference to a satellite contract with the required functionality. As a result, making changes to the functionality of a SC implies creating a new satellite contract and updating its corresponding address in the base contract which will cost less *gas* compared with having all the required functionality in the base contract and having to only update one function before redeployment depending on the size of the base contract. Such design patterns are useful because based on the constructs of the Ethereum blockchain, once deployed, SCs cannot be maintained unlike in the traditional software process where maintenance follows implementation, testing and evolution.

5.1.3 | Number of outgoing invocations

Interestingly, our studied sample of projects did not reveal a significant correlation between CBO (p value = 0.05 andρ = 0.3) and *gasUsed* but revealed a significant correlation with NOI (p# value = 0.0001 andρ = 0.5). Interestingly, the average NOI (p value = 0.001 andρ = 0.3) of all functions in a SC shows a lower correlation to the *gasUsed* metric compared with the count of all outgoing invocations (NOI) of a SC to non-built-in programming language (Solidity) functions.

These results show that CBO does not affect the resources needed to deploy the SCs (i.e., *gasUsed* metric) but the number of calls to methods outside the class has the potential of being an indicator of the *gasUsed* metric. The results provide a practical insight for practitioners with regards to optimising deployment costs for SCs and also provides a statistical background to some existing design patterns for SC development.

Comparison with traditional OO programming

In comparison with traditional software development where CBO has been linked to a high complexity and reduction in reuse, developers can make use of CBO (number of SCs with non-inheritance links to an SC), but on the other hand, they will not need to optimise or minimise the number of calls to built in programming language functionality (e.g., `sha256()`, `require()` and others.)[¶¶¶¶¶] but will need to optimise the number of outgoing calls to functionalities defined in other SCs.

For practitioners

These results are interesting for practitioners because the number of SCs with non-inheritance coupling to an SC does not share a strong link with the deployment costs but the number of outgoing calls to functions defined in other SCs from an SC is important when considering deployment

[§§§§§] More information can be found here: https://github.com/fravoll/solidity-patterns/blob/master/docs/eternal_storage.md
[¶¶¶¶¶] https://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html

TABLE 5 Descriptive statistics of highest correlated metrics for the Token domain

Descriptive statistics					
OO metrics	**Mean**	**Median**	**Mode**	**Min**	**Max**
NOS	16.5	9	12	0	81
DIT	2.5	2	1	0	8
NOA	4.3	2	2	0	12
NOI	6.5	4.5	0	0	28

Abbreviations: DIT, depth of inheritance tree; NOA, number of ancestors; NOI, number of outgoing invocations; NOS, number of statement; OO, objective-oriented.

TABLE 6 Descriptive statistics of highest correlated metrics for the *Others* domain

Descriptive statistics					
OO metrics	**Mean**	**Median**	**Mode**	**Min**	**Max**
NOS	28.7	17	3	2	183
DIT	0.7	0	0	0	3
NOA	1.3	0	0	0	6
NOI	10.9	6	1	1	46

Abbreviations: DIT, depth of inheritance tree; NOA, number of ancestors; NOI, number of outgoing invocations; NOS, number of statement; OO, objective-oriented.

TABLE 7 Spearman's rank correlation of highest correlated metrics across domains and p values ($\alpha = 0.01$)

Spearman's rank correlationρ		
OO metrics	**Tokens**	**Others**
NOS	0.4 ($p = 0.07971$)	**0.5 ($p = 0.00326$)****
DIT	**0.7 ($p = 0.0002$)****	0.4 ($p = 0.00634$)
NOA	**0.7 ($p = 0.0001$)****	0.4 ($p = 0.02041$)
NOI	0.3 ($p = 0.09614$)	**0.5 ($p = 0.00034$)****

Note. Bold emphases indicate strong correlations >= 0.5 and significant where p value <= 0.01>> that can be extended. Abbreviations: DIT, depth of inheritance tree; NOA, number of ancestors; NOI, number of outgoing invocations; NOS, number of statement; OO, objective-oriented.

costs. From a different point of view, we can say that statements with outgoing invocations should be given more attention compared to other statements implemented in a SC as these statements with outgoing invocations form a subset of the NOS metric.

5.2 | Domains (trends in correlated OO metrics and *gasUsed*)

From another point of view, we can also consider the investigated projects by domains. Given the sample of the studied projects, we clustered the projects into two overarching domains: tokens and *others* (covering other decentralised applications such as decentralised insurance, gaming and escrows). This is because majority of the SC projects deployed on the Ethereum blockchain network are oriented towards the creation of a new crypto currency or *alt coin*.[73,74] Four projects from the sample belonged to the token domain, while the other seven were put in the *others* group.

Table 5 shows summary statistics of the correlated metrics in the *Tokens* domain, whereas Table 6 shows summary statistics of the rest of the projects in the *Others* domain. The tables show that although the SCs in the token domain rely more on inherited functionalities (DIT and NOA), the SCs in the others domain are composed of more statements (NOS) and outgoing function invocations (NOI). For more security, certain audited token projects have been created for the purpose of ensuring the security of token-oriented projects as these projects deal with a high volume of funds (equivalent to millions or sometimes billions worth of US dollars[75,76]). During development and before deployment, developers in these domains tend to extend secure and audited programs instead of building theirs from the ground up. Frameworks, such as OpenZeppelin, which are publicly available on GitHub[#####] offers a suite of secure SCs that can be extended.

This is evident by the correlation metrics shown in Table 7. The results in Table 7 are novel, and they demonstrate (statistically significant) large correlations between the inheritance-based metrics (DIT and NOA) and the *gasUsed* metric when considering the *Tokens* domain. On the other hand, we have observed moderate correlations when considering the non-inheritance-based metrics (NOS and NOI) when evaluating the SCs from the seven projects that fall into the *Others* domain in our studied sample.

For practitioners, these results show the existence of trends regarding the correlated metrics across projects from different domains. This can be very useful as it reveals that specific metrics are to be *prioritised* depending on the application domain or goal of the blockchain-oriented

[#####]https://github.com/OpenZeppelin/openzeppelin-contracts

TABLE 8 *Initial* state of the case study Logic.sol smart contract in Figure 3

Smart contract	SLOC	LLOC	CLOC	NF	WMC	NL	NLE	NUMPAR	NOS	DIT	NOA	NOD	CBO	NA	NOI	gasUsed
Logic	12	3	0	1	1	0	0	0	4	0	0	0	1	1	3	234,282

Abbreviations: CBO, coupling between objects; CLOC, comment lines of code; DIT, depth of inheritance tree; LLOC, logical lines of code; NA, number of attributes or states; NF, number of fumctions; NL, nesting level; NLE, nesting level without else-if; NOA, number of ancestors; NOC, number of children; NOD, number of dependencies; NOI, number of outgoing invocation; NOS, number of statement; NUMPAR, number of parameterSLOC, source lines of code; WMC, weighted methods per class.

project when attempting to optimise deployment costs for ethereum blockchain SCs. Furthermore, developers who want to implement IDE (integrated development environment) plugins or tools for optimising gas costs for SC prior to deployment can learn from our empirical results.

5.3 | Case studies (correlation and causation)

Based on the premise that correlation does not always imply causation[64] (given that there could be a third variable), we empirically investigate the causal relationship between the *gasUsed* metric and the moderately correlated source code metrics based on their association with the bytecode of the SCs using the case study or example `Logic.sol` SC shown in Figure 3.

In Section 5.3.1, we investigate the degree to which an increase in the metrics (NOS, DIT, NOA and NOI) with significant correlation affect the size of the bytecode of the SC. Similarly, in Section 5.3.2, we investigate the degree to which an increase in a subset of the metrics (CLOC, NL, NLE, NUMPAR, NOD and CBO) without significant correlation affect the size of the bytecode of the SC.

Prior to investigating the link between the correlated and non-correlated metrics, we need to have a view of the initial state of the SC in Figure 3. Table 8 shows the *initial* state of the SC including the source code metrics and *gasUsed* in its deployment to the Ethereum blockchain network. In addition, the size of the deployed bytecode‖‖‖‖‖‖****** of the SC is initially **596 bytes**.

5.3.1 | Correlated metrics and *gasUsed*

Generally, the SLOC of the Logic.sol SC is 12 (as in Lines 3 to 14 in Figure 3. Focusing on the highest correlated metrics (NOS, DIT, NOA and NOI), Table 8 shows that the initial NOS of the Logic.sol SC is 4 (Lines 7, 10, 11 and 12), whereas the DIT is 0 as the SC is not inheriting functionalities of any contract (as such the NOA is 0). Lastly, the initial NOI is 3 (as in Lines 7, 11 and 12 that make outgoing calls to the DataStorage.sol SC). This is also the reason why the initial CBO is 1 as the Logic.sol SC only shares one non-inheritance relationship with the DataStorage.sol SC and no other SC.

When we replicate Lines 10–12 before redeploying the SC, the NOS increases from 4 to 7, whereas the NOI increases from 3 to 5. The deployed bytecode size in bytes after an increase in both metrics is 1,052 bytes from the initial 596 bytes (difference = 456 bytes). This also causes the *gasUsed* to increase from 234,282 in Table 8 to 350,112 gas (difference = 115,830 gas). This is a significant increase considering that only three lines of code were replicated in the SC.

From these observations, we can deduce that the structural attributes of the SC or the source code metrics (that were found to have the highest significant correlation based on the overall sample of studied projects in Section 4.2) share not just a correlation but also a causal relationship with the *gasUsed* metric via a third variable which is the size of the deployed bytecode in bytes. However, in Section 5.2, we have shown some trends in these metrics when the projects are clustered into domains. As such, we can reject the null hypothesis$H_{2,0}$: the application domains of the SCs do not play a role in the correlations between OO metrics and *gasUsed* but fail to reject the alternative hypothesis$H_{2,1}$: the application domains of the SCs play a role in the correlations between OO metrics and *gasUsed*.

These findings are novel and have an effect on how SC developers can optimise deployment costs based on available resources. Lastly, our results enable developers to control the structural attributes of the source code to optimise the deployment costs as opposed to making changes to the bytecode without knowing how their changes will affect the functionality of the SC.

5.3.2 | Noncorrelated metrics and *gasUsed*

In Section 4.2, we identified some source code metrics with insignificant correlation to the *gasUsed* such as CLOC, NL, NLE, NUMPAR, NOD and CBO. Whereas in Section 5.3.1, we have shown the presence of a causal relationship between the correlated source code metrics and the *gasUsed* by describing how increasing those metrics leads to an increase in the bytecode size of the SC which then has an effect on the *gasUsed* deployment metric. In this section, we will shift our focus to some of the noncorrelated metrics.

Table 8 shows the current state of the SC in Figure 3 including its source code metrics and cost of deployment in terms of *gas*.

When we increase the number of required parameters for the function `f()` by passing both the key and value as function parameters and add four single line comments (two above the constructor and two above the function `f()`) as shown in Figure 9, the CLOC increases as well as the NUMPAR metric of the SC to 2 (two new parameters added to function `f()` in Line 12). The NOD metric remains the same as the SC

‖‖‖‖‖‖ `Example bytecode: 0x6080604052348015610010576000 80fd5b5060405160208061027 8...`
****** One byte is represented by two letters in the bytecode.

```
1  pragma solidity ^0.4.17;
2  import "./DataStorage.sol";
3  contract Logic {
4      DataStorage dataStorage;
5      // this is the constructor
6      // called when smart contract is deployed
7      constructor(address _address) public {
8          dataStorage = DataStorage(_address);
9      }
10     // this function takes in a key
11     // this function takes in a value
12     function f(string kk, uint256 val) public {
13         bytes32 key = keccak256(kk);
14         dataStorage.setUintValue(key, val);
15         dataStorage.getUintValue(key);
16     }
17 }
```

FIGURE 9 Logic.sol smart contract with updated metrics

has no dependants that inherit from it. The SLOC is increased to 15, whereas LLOC is increased to 11. CLOC also increases from 0 to 4 (four commented lines added - Lines 5, 6, 10 and 11).

Upon deployment, the deployed bytecode size in bytes after an increase in most of the noncorrelated metrics with *gasUsed* only shows a minor increase in this case 733 bytes from the initial 596 bytes (difference = 137 bytes). In addition, the *gasUsed* increases from 234,282 in Table 8 to 272,303 gas (difference = 38,021 gas).

If we compare the increases in both the *gasUsed* for deployment and the size of the deployed bytecode of the same Logic.sol SC when the correlated metrics are increased in Section 5.3.1 (such as NOI and NOS) to when we increase the metrics or source code attributes with insignificant correlation in this section, we can observe that the correlated metrics affect the *gasUsed* and bytecode size to a greater degree compared with the noncorrelated metrics such as CLOC and NUMPAR. These observations are significant and not only support the correlation results but also confirm the noncausal relationship between the noncorrelated metrics in Section 4.2 and the *gasUsed* metric.

6 | THREATS TO VALIDITY

6.1 | External validity

This paper presents the results of an empirical analysis that should be applicable to all BOS projects. We cannot generalize our findings to any other sample of open-source software (OSS) projects. Nonetheless, in order to make the findings from our study more generalizable and representative of OSS projects, we have carried out our analysis on a sample of Ethereum blockchain-oriented project hosted on GitHub.[77] The projects also represent different application domains, so the external validity threat is lowered by using this sample. We also acknowledge that the sample size can be small compared with the number of classes studied in traditional OO software research domains. Notwithstanding, in the blockchain domain as demonstrated in the paper, there are costs attached to deploying and invoking artefacts. As such, the number of artefacts (though complex themselves) in BOS projects tends to be smaller compared with larger traditional OO software. In Wu et al,[78] 25 SCs were studied from four BOS projects, whereas 27 were studied in Wüstholz et al.[79]

Furthermore, the scope of our study has been limited to the deployment or gas costs of SCs in relation to their source code metrics. However, we acknowledge that there are other related domains focusing on resource consumption, which we have not explored in this work (e.g., resource estimation in service-oriented environments[80,81] focusing on distributed systems). As an example, the resource metrics used in Kyriazis et al[81] differ from those applicable in the SC domain.[21] Whereas we have focused on the relationship between software metrics such as coupling and inheritance, and gas costs (the resource required for SC deployment), some of the resource or service metrics investigated in Kyriazis et al[81] at the system level include Average Number of Business Processes in the System, Business Processes Capacity of the System, Overall Message Rate in the System (i.e., messages per second), Overall Network Traffic in the System per one unit of time (i.e., bytes per second), Count of simultaneously deployed versions of the services and others.

6.2 | Internal validity

We acknowledge the fact that there could have been some errors in the extraction of the OO metrics from the SCs due to the tools used. To minimize this threat, each metric was manually checked based on their definitions from the literature (as outlined in Section 2.1), in order to mitigate errors. For example, while using the SolMet tool to extract the C&K metrics, we observed a Java programming error in the DIT and NOA computation which was resolved. The AST of some parent SCs were not being parsed before parsing the subcontracts, and this meant that some parent classes were skipped while computing the SC metrics.

Another threat to internal validity we have observed is that other factors may influence *gas* costs. Each low level operation available in the EVM is called an OPCODE. These include operations such as ADD (adding two integers together), BALANCE (getting the balance of an account)

and CREATE (creating a new contract with supplied code to be stored). Each of these OPCODEs has a number called "gas" associated with it. *Gas* is an abstract number that represents the relative complexity of operations. For example, ADD uses 3 *gas* while MUL (multiply two integers) uses 5 *gas*, so MUL is more complex than ADD. Every transaction requires an SC deployment transaction requires a minimum *gas* of 21,000 because all transactions pay this as described in Appendix G (FEE SCHEDULE) in the Ethereum Yellow Paper[1] regarding the$G_{transaction}$ opcode.

Furthermore, deploying a SC requires a minimum of 32,000 *gas*, in addition to 200 *gas* per byte of the compiled source code, as described in Appendix G (FEE SCHEDULE) in the Ethereum Yellow Paper[1] regarding the CreateG_{create} and CodeDeposit$G_{codedeposit}$ opcodes. Deducting the constant 53,000 (32,000 and 21,000) gas from the *gasUsed* for all the studied SCs will also not alter the correlation results. In addition, as described in Section 1, the rationale for size metrics comes in here because the bytecode of the SC cannot be properly adjusted or shortened to reduce deployment costs while maintaining the required functionality of the SC. Reducing the size of the contract has to be done prior to compilation (or conversion to bytecode) via the source code that is more understandable to developers.

Lastly, some of the analysed projects have a small number of SCs and might not add meaning to the correlation results. For example, the CBO metric is 0 for the *Kleros* project as only one contract is being deployed in the project as of the time of the study. The project does not make use of some of the design patterns for SCs as discussed in Wöhrer and Zdun[72] compared with the Gnosis project that uses the Oracle (data provider) pattern and the Data Segregation pattern and as such has SCs with CBO> 0.

6.3 | Construct validity

The scope of our sample of projects was limited to SCs written in the Solidity programming language for the Ethereum blockchain. Ethereum is a public blockchain platform that requires the use of *gas* resources to use most of the functionalities of SCs. Other SC-based blockchain platforms exist, such as Hyperledger, which uses SCs written in Golang. However, these SCs do not require any resources to deploy and use: Hyperledger is a private blockchain platform and does not require the payment of miner nodes for transaction approval and inclusion in blocks. As a second threat to contruct validity, the Spearman'sρ was used to assess the correlation between the metrics and the gas costs. Although the test has been widely used in past research, it also has its disadvantages: it takes into consideration the ranked order of the values (OO metrics, e.g., CBO and `gasUsed`) and not the values themselves. In other words, if the order of the values is the same, the coefficient will stay the same.

7 | RELATED WORK

In this section, we provide an overview of related studies that have considered the structural metrics or architecture of blockchain-oriented software.

The initial study on SC metrics was performed by Tonelli et al.[43] The researchers studied SCs software metrics†††††† extracted from a set of SCs deployed on the public Ethereum blockchain network with the goal of finding out if given the uniqueness of SC software development, the corresponding software metrics will show differences in statistical properties with respect to metrics derived from traditional software systems (e.g., Java source code metrics). For each software metric, the researchers computed standard statistics like average, median, maximal and minimal values, and standard deviation. The study was based on the assumptions that resources are limited on the blockchain and such limitations may influence the way SCs are written. Their metrics were based on source code as well as bytecode of SCs, but with regards to source code metrics, the authors only analysed SLOC. The authors did not investigate metrics such as inheritance or the other C&K metrics such as CBO or DIT as done in this study.

Similarly, Hegedűs has investigated the nature of the typical structure of SCs with regards to structural metrics.[21] A tool called SolMet was developed to extract the size, complexity, coupling and inheritance metrics from a range of SCs already deployed to the Ethereum livenet. In general, the results revealed almost all typical metrics in the context of SCs have lower values compared with OO programmes. The metrics derived in this study are also very low compared with OO software, for example, the NOC metric. Deployed SCs are more reliant on parent contracts but their features are seldom inherited.

Ducasse et al[20] state, "Due to the extremely fast growing pace of SC usage, in this new software paradigm measuring code quality is becoming as essential as in out-of-chain software development". The authors mentioned as future work the development of a tool to capture metrics and that traditional metrics are not sufficient for evaluating SCs. However, this has not been demonstrated in an empirical study. They further emphasized the need for *gas* estimation tools. In this study, we have empirically addressed both concerns: investigating traditional metrics in the context of SCs and investigating their correlation with *gas* costs.

In a related study, Vandenbogaerde[56] proposed a graph-based framework for computing design metrics for SCs from an OO point of view (inspired by Mens and Lanza[82]) and applied the framework in a preliminary study. The implemented framework allows the use of simple queries to extract functions and design metrics from the generated graph-based semantic meta-model, for example, number of function calls for all SCs in a project as shown in the example in the study: `g.V().contract().functions().isCalled().count()`. The calculated design

†††††† The metrics studied included total lines of code associated with a specific SC at a blockchain address, the number of SCs inside a single address code (this is analogous of classes in Java files, e.g., compilation units), blank lines, comment lines, number of static calls to events, number of modifiers, number of functions, number of payable functions, cyclomatic complexity as the simplest McCabe definition,[36] number of mappings to addresses and the size of the associated bytecode and of the vector of contracts' ABIs.

metrics include cyclomatic complexity, number of lines, number of functions, depth of inheritance tree, and others. In contrast, in our study, we have investigated more metrics extracted from the SCs using the SolMet tool[21] used in a similar work to the study by Vandenbogaerde.[56] The author mentions the development of design metrics that are specific to SCs as part of future work. The author also mentions that the inheritance mechanisms that the Solidity smart programming language provides seems to be underutilised as inheritance trees are not deep, and on average, a contract does not have many children. We have identified a similar pattern in our study and in contrast, we have further shown how inheritance metrics show the strongest link to *gas* or deployment costs for SCs using correlation analysis.

According to Wessling and Gruhn[83] "building blockchain-oriented applications forces developers to rethink the architecture of their software from the ground up". The researchers explored decentralised applications and their architecture with the goal of finding reoccurring architectural patterns and their impacts on security and trust. Their work provides insights into architectural patterns for blockchain-oriented software applications and provides a rationale regarding why it is necessary for developers to think of how users will make use of decentralised applications.

Lastly, Zhang *et al.*,[7] provided evaluation metrics that can be used to examine blockchain-based decentralised applications with regards to their feasibility, intended capability and compliance in the healthcare domain‡‡‡‡‡‡. However they did not perform an empirical study using the proposed metrics and have not proposed structural software or source code metrics in relation to gas resources consumed by Ethereum blockchain transactions.

8 | CONCLUSION AND FURTHER WORK

Prior research has emphasised the need for effective software development in decentralised application contexts[6] and the need for automating the metrics extraction to measure the quality of SCs. In this study, we have carried out a novel empirical analysis on the relationship between traditional OO software metrics and the actual resources consumed when deploying SCs on the Ethereum blockchain network.

Results from this study have revealed statistically significant and strong correlation between some of the *inheritance-based* OO metrics (DIT and NOA) investigated and the resources required for SC deployment, but insignificant correlation with *non-inheritance* -related coupling metrics such as CBO. We have also discussed the relationship with the observed results and SC design patterns. It is also noteworthy that we observed trends in the correlated metrics when the blockchain-oriented projects are clustered into application domains in Section 5.2 showing specific metrics to be given more priority based on the application domain a project belongs to and we explored the causal relationship between both the metrics that shared a significant and insignificant correlation to the SC deployment costs in Sections 5.3.1 and 5.3.2, which supported the initial correlation results. We identified that compared with the metrics with an insignificant correlation, the metrics with a statistically significant and moderate to large correlation to the deployment resources have a larger direct impact on the size of the deployed bytecode of the SC that also influences the deployment costs.

These results are significant and will have an impact in SC development practices. At a higher level, the results will guide practitioners about the structural changes or refactorings that could be made in order to minimise deployment resources. These refactorings can also be semi-automated in the form of SC development tools learning from our results. Finally, for SC developers, the metrics extracted from the contracts will be useful to inform the amount of *gas* that they will be able to devote for the execution of the SC.

Future work will include the analysis of design patterns for SCs and resource usage at the function level. We also aim to investigate automated testing in the context of SCs (e.g., flaky tests and mutation testing) to minimise bugs post-deployment given that the nature of the Ethereum blockchain does not permit SC updates or modifications post-deployment. Library recommendation techniques for secure and reliable SC development also seems feasible.

ORCID

Nemitari Ajienka https://orcid.org/0000-0002-8792-282X

REFERENCES

1. Wood G. Ethereum yellow paper. Internet: https://github.com/ethereum/yellowpaper,[Oct. 30, 2018]; 2014.
2. Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. www.bitcoin.org; 2008.
3. Buterin V. Ethereum white paper: a next generation smart contract & decentralized application platform. ethereum.org; 2013.
4. Gervais A, Karame GO, Wüst K, Glykantzis V, Ritzdorf H, Capkun S. On the security and performance of proof of work blockchains. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna, Austria: ACM; 2016:3-16.
5. Bartoletti M, Pompianu L. An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. In: International Conference on Financial Cryptography and Data Security. Sliema, Malta: Springer; 2017:494-509.
6. Porru S, Pinna A, Marchesi M, Tonelli R. Blockchain-oriented software engineering: challenges and new directions. In: 39th IEEE/ACM International Conference on Software Engineering Companion. Buenos Aires, Argentina: (ICSE-C); 2017:169-171.

‡‡‡‡‡‡The authors have provided metrics such as support for user identification and authentication, support for structural interoperability at minimum, scalability across large populations of healthcare participants, cost-effectiveness and support of patient-centred care model.

7. Zhang P, Walker MA, White J, Schmidt DC, Lenz G. Metrics for assessing blockchain-based healthcare decentralized apps. In: 19th IEEE International Conference on E-Health Networking, Applications and Services. Dalian, China: (HEALTHCOM); 2017:1-4.
8. Luu L, Teutsch J, Kulkarni R, Saxena P. Demystifying incentives in the consensus computer. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer And Communications Security. Denver, USA; 2015:706-719.
9. Salah K, Rehman MHU, Nizamuddin N, Al-Fuqaha A. Blockchain for AI: review and open research challenges. *IEEE Access*. 2019;7:10127-10149.
10. Kolluri A, Nikolic I, Sergey I, Hobor A, Saxena P. Exploiting the laws of order in smart contracts. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis ACM. Beijing, China; 2019:363-373.
11. Liu C, Liu H, Cao Z, Chen Z, Chen B, Roscoe B. Reguard: finding reentrancy bugs in smart contracts. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings ACM. Gothenburg Sweden; 2018:65-68.
12. Jiang B, Liu Y, Chan W. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering ACM. Montpellier, France; 2018:259-269.
13. Mavridou A, Laszka A. Designing secure ethereum smart contracts: a finite state machine based approach. arXiv preprint arXiv:171109327; 2017.
14. Chatterjee K, Goharshady AK, Velner Y. Quantitative analysis of smart contracts. *European Symposium on Programming*. Cham: Springer; 2018:739-767.
15. Atzei N, Bartoletti M, Cimoli T. A survey of attacks on ethereum smart contracts (sok). In: International Conference on Principles of Security and Trust. Uppsala, Sweden: Springer; 2017:164-186.
16. Androulaki E, Barger A, Bortnikov V, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth Eurosys Conference. Porto Portugal: ACM; 2018:30.
17. Hearn M, Brown RG. Corda: A distributed ledger; 2016.
18. Valenta M, Sandner P. Comparison of ethereum, hyperledger fabric and corda. [ebook] Frankfurt School, Blockchain Center; 2017.
19. Coulin T, Detante M, Mouchère W., Petrillo F. Software architecture metrics: a literature review. arXiv preprint arXiv:190109050; 2019.
20. Ducasse S, Rocha H, Bragagnolo S, Denker M, Francomme C. SmartAnvil: Open-source tool suite for smart contract analysis. In: Ragnedda M, Destefanis G, eds. *Blockchain and web 3.0: Social, economic, and technological challenges*: Routledge; 2019.
21. Hegedűs P. Towards analyzing the complexity landscape of solidity based ethereum smart contracts. *Technologies*. 2019;7(1):6.
22. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Trans Softw Eng*. 1994;20(6):476-493.
23. Aloysius A, Arockiam L. Coupling complexity metric: a cognitive approach. *Int J Infor Technol Comput Sci*. 2012;4(9):29-35.
24. Briand LC, Daly JW, Wüst JK. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans Softw Eng*. 1999;25(1):91-121.
25. Kitchenham B. What's up with software metrics?–a preliminary mapping study. *J Syst Softw*. 2010;83(1):37-51.
26. Li W, Henry S. Object-oriented metrics that predict maintainability. *J Syst Softw*. 1993;23(2):111-122.
27. Oliva GA, Gerosa MA. On the interplay between structural and logical dependencies in open-source software. In: 25th Brazilian Symposium on Software Engineering (SBES). Sao Paulo, Brazil; 2011:144-153.
28. Chhikara A, Chhillar R, Khatri S. Evaluating the impact of different types of inheritance on the object oriented software metrics. *Int J Enterpr Comput Business Syst*. 2011;1(2):1-7.
29. Counsell S, Mendes E, Swift S. Comprehension of object-oriented software cohesion: the empirical quagmire. In: Proceedings of the 10th International Workshop on Program Comprehension (IWPC). Paris, France; 2002:33-42.
30. D'Ambros M, Lanza M, Robbes R. An extensive comparison of bug prediction approaches. In: 7th IEEE Working Conference on Mining Software Repositories (MSR). Cape Town, South Africa; 2010:31-41.
31. El Emam K, Melo W, Machado JC. The prediction of faulty classes using object-oriented design metrics. *J Syst Softw*. 2001;56(1):63-75.
32. Radjenović D., Heričko M, Torkar R, Živkovič A. Software fault prediction metrics: a systematic literature review. *Inform Softw Technol*. 2013;55(8):1397-1418.
33. Capretz LF, Xu J. An empirical validation of object-oriented design metrics for fault prediction. *J Comput Sci*. 2008;4(7):571.
34. Basili VR, Briand LC, Melo WL. A validation of object-oriented design metrics as quality indicators. *IEEE Trans Softw Eng*. 1996;22(10):751-761.
35. Wilkie FG, Kitchenham BA. Coupling measures and change ripples in C++ application software. *J Syst Softw*. 2000;52(2-3):157-164.
36. McCabe TJ. A complexity measure. *IEEE Trans Softw Eng*. 1976;SE-2(4):308-320.
37. Dannen C. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners* (Vol. 1): Berkeley: Apress: Springer; 2017.
38. Ebert C, Cain J. Cyclomatic complexity. *IEEE Softw*. 2016;33(6):27-29.
39. Destefanis G, Marchesi M, Ortu M, Tonelli R, Bracciali A, Hierons R. Smart contracts vulnerabilities: a call for blockchain software engineering? In: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). Campobasso, Italy: IEEE; 2018:19-25.
40. Eyal I, Gencer AE, Sirer EG, Van Renesse R. Bitcoin-NG: a scalable blockchain protocol. In: 13th Usenix Symposium on Networked Systems Design and Implementation. Santa Clara, CA, United States; 2016:45-59.
41. Wohrer M, Zdun U. Smart contracts: security patterns in the ethereum ecosystem and solidity. In: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). Campobasso, Italy: IEEE; 2018:2-8.
42. Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y. Smartcheck: static analysis of ethereum smart contracts. In: 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering For Blockchain (WETSEB). Gothenburg, Sweden: IEEE; 2018:9-16.
43. Tonelli R, Destefanis G, Marchesi M, Ortu M. Smart contracts software metrics: a first study. arXiv preprint arXiv:180201517; 2018.
44. Luu L, Chu D-H, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security ACM. Vienna, Austria; 2016:254-269.
45. Zhang P, White J, Schmidt DC, Lenz G. Applying software patterns to address interoperability in blockchain-based healthcare apps. arXiv preprint arXiv:170603700; 2017.
46. Grech N, Kong M, Jurisevic A, Brent L, Scholz B, Smaragdakis Y. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *Proc ACM Program Lang*. 2018;2(OOPSLA):116:1-27.

47. Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D. The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories ACM. Hyderabad India; 2014:92-101.
48. Vasilescu B, Serebrenik A, Filkov V. A data set for social diversity studies of github teams. In: Proceedings of the 12th Working Conference On Mining Software Repositories. Florence, Italy: IEEE Press; 2015:514-517.
49. Kikas R, Dumas M, Pfahl D. Using dynamic and contextual features to predict issue lifetime in github projects. In: Proceedings of the 13th International Conference on Mining Software Repositories ACM. Austin, TX, USA; 2016:291-302.
50. Abdalkareem R, Nourry O, Wehaibi S, Mujahid S, Shihab E. Why do developers use trivial packages? An empirical case study on NPM. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Paderborn, Germany: ACM; 2017:385-395.
51. Lin B, Nagy C, Bavota G, Lanza M. On the impact of refactoring operations on code naturalness. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). Hangzhou, China: IEEE; 2019:594-598.
52. Crowston K, Annabi H, Howison J. Defining open source software project success. In: ICIS 2003 Proceedings. Seattle, Washington; 2003:28.
53. Borges H, Valente MT, Hora A, Coelho J. On the popularity of github applications: a preliminary note. arXiv preprint arXiv:150700604; 2015.
54. Alshomali MA, Hamilton JR, Holdsworth J, Tee S. Github: factors influencing project activity levels. In: Proceedings 17th international conference on electronic business. Dubai, United Arab Emirates; 2017:295-303.
55. Borges HS, Valente MT. How do developers promote open source projects? *Computer*. 2019;52(8):27-33.
56. Vandenbogaerde B. A graph-based framework for analysing the design of smart contracts, ESEC/FSE 2019. ACM; 2019; New York, NY, USA:1220-1222. http://doi.acm.org/10.1145/3338906.3342495
57. Norick B, Krohn J, Howard E, Welna B, Izurieta C. Effects of the number of developers on code quality in open source software: a case study. In: *ESEM ESEM ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. Bolzano-Bozen Italy; 2010:1-1.
58. Constantinou E, Kakarontzas G, Stamelos I. Towards open source software system architecture recovery using design metrics. In: 15th Panhellenic Conference on Informatics. Kastoria, Greece: IEEE; 2011:166-170.
59. Fenton N, Bieman J. *Software Metrics: A Rigorous and Practical Approach*. Vancouver: CRC Press; 2014.
60. Yu L. Understanding component co-evolution with a study on Linux. *Empir Softw Eng*. 2007;12(2):123-141.
61. Pagano RR. *Understanding Statistics in the Behavioral Sciences* (Vol. 1). 6th ed.: Vancouver: Wadsworth-Thomson Learning; 2001.
62. Field AP. *Discovering statistics using IBM SPSS statistics*. SAGE; 2013.
63. Marcus A, Poshyvanyk D. The conceptual cohesion of classes. In: 21st IEEE International Conference on Software Maintenance (ICSM'05) Budapest, Hungary: IEEE; 2005:133-142.
64. Aggarwal R, Ranganathan P. Common pitfalls in statistical analysis: the use of correlation techniques. *Perspect Clin Res*. 2016;7(4):187.
65. Elish KO, Alshayeb M. A classification of refactoring methods based on software quality attributes. *Arab J Sci Eng*. 2011;36(7):1253-1267.
66. Bavota G, De Lucia A, Di Penta M, Oliveto R, Palomba F. An experimental investigation on the innate relationship between quality and refactoring. *J Syst Softw*. 2015;107:1-14.
67. Aggarwal K, Singh Y, Kaur A, Malhotra R. Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study. *Softw Process: Improv Pract*. 2009;14(1):39-62.
68. Khalid S, Zehra S, Arif F. Analysis of object oriented complexity and testability using object oriented design metrics. In: Proceedings of the 2010 National Software Engineering Conference. Rawalpindi, Pakistan: ACM; 2010:4.
69. Shatnawi R, Li W. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *J Syst Softw*. 2008;81(11):1868-1882.
70. Thongmak M, Muenchaisri P. Predicting faulty classes using design metrics with discriminant analysis; Software engineering research and practice. 2003:621-627.
71. Liu Y, Lu Q, Xu X, Zhu L, Yao H. Applying design patterns in smart contracts. In: International Conference On Blockchain. Halifax, Canada: Springer; 2018:92-106.
72. Wöhrer M, Zdun U. Design patterns for smart contracts in the ethereum ecosystem. In: 2018 IEEE International Conference on Internet of things (ITHINGS) and IEEE Green Computing and Communications (GREENCOM) and IEEE cyber, Physical and Social Computing (CPSCOM) and IEEE Smart Data (SMARTDATA). Halifax, NS, Canada: IEEE; 2018:1513-1520.
73. Cagli EC. Explosive behavior in the prices of bitcoin and altcoins. *Finance Res Lett*. 2019;29:398-403.
74. Yuryev G. What can explain the performance of initial coin offerings? University of Stavanger. *Master's Thesis*; 2018.
75. Feng Q, He D, Zeadally S, Khan MK, Kumar N. A survey on privacy protection in blockchain system. *J Netw Comput Appl*. 2018;126:45–58.
76. Hargrave J, Sahdev N, Feldmeier O, et al. How value is created in tokenized assets. In: Melanie, S, Jason, P, Soichiro, T, Frank, W and Paolo, T. eds. *Blockchain Economics: Implications of Distributed Ledgers-Markets, Communications Networks, and Algorithmic Reality*. (Vol. 1): World Scientific; 2019.
77. Feldt R, Magazinius A. Validity threats in empirical software engineering research— an initial survey. In: 22nd International Conference on Software Engineering and Knowledge Engineering. Pittsburgh, USA: (SEKE); 2010:374-379.
78. Wu H, Wang X, Xu J, Zou W, Zhang L, Chen Z. Mutation testing for ethereum smart contract. arXiv preprint arXiv:190803707; 2019.
79. Wüstholz V., Christakis M. Harvey: a greybox fuzzer for smart contracts. arXiv preprint arXiv:190506944; 2019.
80. Kyriazis D, Menychtas A, Kousiouris G, et al. A real-time service oriented infrastructure; 2010.
81. Rud D, Schmietendorf A, Dumke R. Resource metrics for service-oriented infrastructures. In: Proc SEMSOA 2007. Hannover, Germany; 2007:90-98.
82. Mens T, Lanza M. A graph-based metamodel for object.-oriented software metrics. *Electron Notes Theoret Comput Sci*. 2002;72(2):57-68.
83. Wessling F, Gruhn V. Engineering software architectures of blockchain-oriented applications. In: 2018 IEEE International conference on software architecture companion (icsa-c). Seattle, Washington; 2018:45-46.

How to cite this article: Ajienka N, Vangorp P, Capiluppi A. An empirical analysis of source code metrics and smart contract resource consumption. *J Softw Evol Proc*. 2020;32:e2267. https://doi.org/10.1002/smr.2267